Adaptation d'un algorithme d'ordonnancement de tâches parallèles sur plates-formes homogènes aux systèmes hétérogènes

Tchimou N'Takpé

École Doctorale IAEM Lorraine /
DEA Informatique de Lorraine

Institut National
Polytechnique de Lorraine

ALGORILLE / LORIA

Adaptation d'un algorithme d'ordonnancement de tâches parallèles sur plates-formes homogènes aux systèmes hétérogènes

Tuteur : Frédéric SUTER

Étudiant : Tchimou N'TAKPÉ

Date de soutenance : 22 juin 2005

Jury :
- D. Méry
- D. Galmiche
- N. Carbonell
- O. Festor
- F. Suter

Remerciements

Je tiens à remercier toute l'équipe du projet ALGORILLE, notamment son directeur M. Jens Gustedt ainsi que MM. Emmanuel Jeannot, Martin Quinson et Frédéric Wagner qui ont tous eu un regard critique au cours du déroulement de ce stage.

Je ne saurai oublier mon tuteur, M. Frédéric Suter, pour sa disponibilité et sans qui les simulations nécessaires à la conclusion de cette étude n'auraient pas pu se dérouler.

Table des matières

Introduction

La mutualisation des ressources informatiques à travers les grilles de calcul permet aujourd'hui d'accéder à moindre coût à des supercalculateurs via l'Internet ou au sein de certaines entreprises. Les applications qui ciblent ces grilles de calcul sont nombreuses et la plupart d'entre elles sont très gourmandes en calcul et en capacité de stockage. Ainsi ces grilles de calcul mettent à leur disposition de nombreuses ressources de calcul et de stockage reliées par un réseau haut débit.

De nombreuses activités de recherches menées ces dernières années ont permis de développer non seulement des intergiciels adaptés aux grilles, mais aussi des modèles de tâches capables de s'exécuter sur plusieurs processeurs en même temps, les *tâches parallèles*. Ces tâches utilisent le paradigme du *parallélisme de données* qui consiste à appliquer la même opération en parallèle sur différents éléments d'un ensemble de données, ce qui permet normalement de diminuer leur temps d'exécution. Dans l'optique de diminuer les temps d'exécution des grosses applications sur les grilles de calcul, il est également possible d'utiliser le paradigme du *parallélisme de tâches* défini par l'exécution concurrente de calculs distincts sur des ensembles de données différents. L'exploitation simultanée de ces deux paradigmes a plusieurs avantages dont celui d'augmenter l'extensibilité d'un programme en exhibant plus de parallélisme.

La plupart des études effectuées sur l'ordonnancement des tâches parallèles ne traitent que du cas de plates-formes homogènes. L'approche la plus utilisée consiste à découpler la phase d'allocation de processeurs de celle de l'ordonnancement. La phase d'allocation permet d'allouer des processeurs à chaque tâche de manière à obtenir un compromis entre l'utilisation des processeurs et le temps d'exécution de l'application. La phase d'ordonnancement sert à placer les tâches sur les processeurs disponibles généralement à l'aide d'un algorithme d'ordonnancement de liste.

Or, les grilles de calculs se font de plus en plus nombreuses et sont en général constituées par l'agrégation hétérogène de plusieurs grappes homogènes. L'objectif de se stage est donc de concevoir et d'évaluer un algorithme d'ordonnancement de tâches parallèles en milieu hétérogène en partant d'un de ces algorithmes en deux étapes. Cet algorithme va permettre de mutualiser les ressources des plates-formes homogènes existantes pour l'exécution d'applications de grandes tailles.

La première section (section 1) de ce rapport présente les modèles de plates-formes, de tâches et d'applications que nous utiliserons dans notre étude.

La section 2 est un état de l'art des différents travaux existant dans le domaine de l'ordonnancement de tâches parallèles.

Ceci nous amène dans la section 3, à la description plus détaillée de l'algorithme d'ordonnancement de tâches parallèles en milieu homogène que nous avons choisi pour être adapté aux plates-formes hétérogènes.

La section 4 présente comment nous passons des plates-formes homogènes aux plates-formes hétérogènes à partir de l'algorithme choisi. À l'issue de cette étape de l'étude, nous aboutissons à deux algorithmes qui se distinguent par les critères de priorité ayant pour but de discriminer les tâches prêtes pendant la phase d'ordonnancement.

Dans la section 5, il sera question de décrire le plan de test que nous avons mis en place pour évaluer les deux nouveaux algorithmes.

Enfin, avant de conclure notre étude, la dernière section (section 6) sera consacrée à une étude comparative entre les algorithmes que nous avons mis en œuvre et divers autres algorithmes d'ordonnancement de tâches parallèles dont celui qui nous a servi de point de départ.

1 Modèles utilisés

Cette section définit les modèles de plate-forme, de tâche et d'application que nous adopterons dans cette étude.

1.1 Modèle de plate-forme

Les grilles que nous allons utiliser (voir figure 1) sont constituées d'une ou plusieurs grappes. Chaque grappe est constituée d'un ensemble homogène de processeurs reliés entre eux par le biais d'un commutateur (Switch). Les grappes communiqueront entre elles en étant reliées par le biais d'une passerelle (Gateway) à un réseau très haut débit (Backbone). La capacité des passerelles limite le nombre de processeurs qui peuvent émettre en même temps sur le réseau.

Cette plate-forme correspondont à la notion de grille légère introduite dans [1] dans laquelle on retrouve des sous-ensembles homogènes de processeurs.

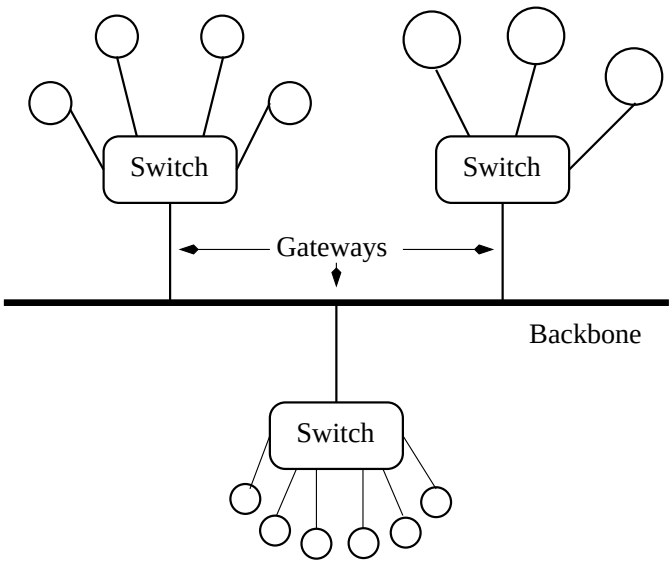


FIG. 1 – Description de la plate-forme

1.2 Modèle de tâche

De manière informelle, une tâche parallèle est une tâche qui contient des opérations élémentaires, typiquement une routine numérique ou des boucles imbriquées, qui contiennent elles-mêmes suffisamment de parallélisme pour être exécutées par plus d'un processeur. Il est possible de distinguer trois classes de tâches parallèles :

- Les tâches **rigides** où le nombre de processeurs exécutant la tâche parallèle est fixé *à priori*. Dans ce cas, la tâche peut être représentée par un rectangle dans un diagramme de Gantt.
- Les tâches **modelables** pour lesquelles le nombre de processeurs n'est pas fixé mais déterminé avant l'exécution. Cependant, comme dans le cas précédent, ce nombre de processeurs ne change pas jusqu'à la fin de l'exécution.
- Les tâches **malléables** peuvent voir le nombre de processeurs qui leur est alloué changer au cours de l'exécution (par préemption des tâches ou par redistribution de données).

Pour des raisons historiques, la plupart des applications sont traitées comme étant composées de tâches rigides. Pourtant la plupart des applications parallèles sont intrinsèquement modelables puisqu'un programmeur ne connaît pas à l'avance le nombre exact de processeurs qui seront utilisés à l'exécution. Le caractère malléable d'une tâche est plus facilement utilisable du point de vue de l'ordonnancement mais requiert des fonctionnalités avancées de la part de l'environnement d'exécution rarement disponibles.

Dans cette étude, nous ferons l'hypothèse que les tâches utilisées sont modelables. Ceci nous amène à adopter le modèle d'application décrit dans la section suivante.

1.3 Modèle d'application

Une application parallèle peut être modélisée par un graphe acyclique orienté $G = (\nu, \epsilon)$ où ν est un ensemble de V nœuds et ϵ est un ensemble de E arêtes. Les nœuds représentent les tâches parallèles et les arêtes définissent les relations de précédence (dépendances de flots ou de données) entre les différentes tâches. Ces tâches parallèles sont des tâches non préemptives capables de s'exécuter sur un nombre quelconque de processeurs.

On appellera *tâche d'entrée* du graphe, une tâche n'ayant aucun prédécesseur et *tâche de sortie*, une tâche n'ayant aucun successeur. Une *tâche prête* est une tâche dont tous les prédécesseurs ont terminés leur exécution.

Dans un système homogène de processeurs, le temps d'exécution d'une tâche parallèle $t \in \nu$ peut être modélisé par la loi d'Amdahl :

$$T_\omega(t, N_p(t)) = \left(\alpha + \frac{1-\alpha}{N_p(t)}\right) \cdot T_\omega(t, 1)$$

où $N_p(t)$ est le nombre de processeurs alloués à t, $T_\omega(t, 1)$ le temps d'exécution de la même tâche sur un seul processeur et α la portion non parallélisable de la tâche.

On associe alors $T_\omega(t, N_p(t))$ à chaque nœud t du graphe de tâches et on définit :

- **le *top level*** comme étant le chemin le plus long depuis une *tâche d'entrée* quelconque jusqu'à la tâche t excluant le temps d'exécution de t.
- **le *bottom level*** comme étant chemin le plus long depuis la tâche t incluant le temps d'exécution de t jusqu'à une *tâche de sortie* quelconque.

Dans la section qui suit, nous présenterons diverses approches pragmatiques de l'ordonnancement de tâches parallèles de la littérature.

2 Travaux précédents

L'ordonnancement des tâches étant un problème NP-complet, même dans le cas des tâches séquentielles, beaucoup d'heuristiques ont été développées pour l'ordonnancement des tâches parallèles. La quasi totalité de ces heuristiques ne traitent que de l'ordonnancement de tâches parallèles en milieux homogènes. Cette section présente l'essentiel des algorithmes proposés.

Les travaux de Ramaswamy *et al.* [2] introduisent la notion de *Macro Dataflow Graph* (MDG) qui permet de décrire les applications parallèles et présentent un algorithme d'ordonnancement en deux étapes. Un MDG est un graphe acyclique orienté où les nœuds représentent des calculs séquentiels ou parallèles et les arcs représentent les relations de précédence (voir section 1.3). L'algorithme d'ordonnancement proposé par Ramaswamy utilise la programmation convexe, rendue possible grâce

à la propriété de *posynomialité* des modèles de coût choisis, ainsi que certaines propriétés du MDG. Le chemin critique est défini comme étant le plus long chemin dans le MDG et le temps de complétion associé est donc minimal. L'autre métrique utilisée est l'aire moyenne, définie comme le produit *temps* $\times$ *nombre de processeurs* moyen du MDG. L'algorithme TSAS (*Two Step Allocation and Scheduling*) se décompose en deux étapes. La première cherche à minimiser le temps de complétion selon les deux métriques présentées ci-dessus. Cette étape permet de déterminer le placement de chacune des tâches du graphe. La seconde étape est basée sur un algorithme par liste pour ordonnancer les tâches ainsi placées.

Rădulescu *et al.* ont également proposé deux algorithmes d'ordonnancement de tâches parallèles en deux étapes : CPR (*Critical Path Reduction*) [3] et CPA (*Critical Path and Area-based scheduling*) [4]. Tous deux sont basés sur la réduction du chemin critique de l'application. La principale différence entre CPR et CPA est que le processus d'allocation est complètement découplé de l'ordonnancement dans CPA. Dans l'étape d'allocation, ces deux algorithmes cherchent à déterminer le nombre de processeurs le plus approprié à l'exécution de chacune des tâches. Pour cela, ils allouent tout d'abord un seul processeur à chacune des tâches. Puis, pour chaque tâche, des processeurs supplémentaires sont ajoutés un par un, tant que le temps d'exécution de cette tâche décroît, conformément à la loi d'Amdahl, et que le nombre total de processeurs disponibles n'est pas atteint. La seconde étape ordonne les tâches ainsi placées selon un algorithme par liste, comme pour TSAS.

Rauber et Rünger [5] limitent quant à eux leur étude à des graphes construits par compositions séries et/ou parallèles. Dans le premier cas, une séquence d'opérations présentant des dépendances de données est placée sur l'ensemble des processeurs. Les tâches de cette séquence sont alors exécutées séquentiellement. Dans le second cas, l'ensemble des processeurs est divisé en un nombre optimal de sous-ensembles, déterminé par un algorithme glouton. Le critère d'optimalité de cet algorithme est la minimisation du temps de complétion de l'ensemble de tâches considéré. Les temps d'exécution des routines parallèles sont estimés par des formules dépendant des coûts de communication et des temps correspondants à des exécutions séquentielles.

Boudet, Desprez et Suter ont mis en place un algorithme d'ordonnancement mixte à étape unique et sans réplication des données [6]. Il est basé sur une estimation précise de chacune des tâches en fonction de la plate-forme utilisée. Cet algorithme ne fait aucune hypothèse sur l'homogénéité de la plate-forme. chaque tâche s'exécutera sur l'un des sous-ensembles homogènes de processeurs prédéfinis de la plate-forme.

Des travaux précédemment réalisés par Casanova, Desprez et Suter [7] ont enfin permis d'adapter des heuristiques d'ordonnancement par liste de tâches séquentielles sur plates-formes hétérogènes au cas des tâches parallèles. Ces travaux constituent une première approche pour l'ordonnancement des tâches parallèles sur les plates-formes hétérogènes.

Notre objectif est d'introduire l'hétérogénéité dans un algorithme d'ordonnancement de tâches mixtes sur plates-formes homogènes. Pour cela nous avons retenu CPA qui est le plus performant des algorithmes en deux étapes que nous venons de citer. Les résultats de notre algorithme seront confrontés à ceux de l'approche précédente ainsi qu'à l'algorithme d'origine.

Nous allons maintenant présenter le principe de l'algorithme CPA qui servira de point de départ pour cette étude.

3 Présentation de l'algorithme CPA

Le tableau Tab. 1 présente les différentes notations qui seront utilisées dans la présentation de l'heuristique CPA.

T_t	top level de la tâche t
T_b	bottom level de la tâche t
T_{CP}	chemin critique du graphe de tâches
T_A	temps moyen d'occupation des processeurs
$T_s(t)$	date de début d'exécution d'une tâche t
$T_f(t)$	date de fin d'exécution d'une tâche t

Tab. 1 – Liste des notations utilisées.

CPA (Critical Path and Area-based Scheduling) [4] est un algorithme d'ordonnancement de tâches parallèles en deux étapes. Dans la première étape (MA), on détermine le nombre de processeurs $N_p(t)$ à allouer à chaque tâche t. La seconde étape (MLS) consiste à ordonnancer les tâches grâce à un algorithme de liste.

La plate-forme utilisée est composée de P processeurs homogènes et on a toujours $N_p(t) \leq P$.

Avec une complexité de l'ordre de $O(V(V+E)P)$, les auteurs de l'heuristique CPA démontrent qu'elle est plus performante que TSAS [2], TWOL [5] et CPR [3] pour l'ordonnancement des tâches parallèles sur plates-formes homogènes.

Le paragraphe qui va suivre présente la phase d'allocation de CPA.

3.1 Allocation de processeurs

Le temps d'exécution d'une application parallèle peut être approximé par :

$$T_p^e = \max\{T_{CP}, T_A\},$$

où T_{CP} est le chemin critique du graphe de tâches de l'application et T_A, le temps moyen d'occupation des processeurs qui découle du diagramme de Gantt de l'application. On a :

$$T_{CP} = \max_{t \in \nu} T_b(t),$$

$$T_A = \frac{1}{P} \sum_{t \in \nu} \left(T_\omega \left(t, N_p(t) \right) \times N_p(t) \right).$$

Le but de CPA est de minimiser T_p^e au terme de cette phase d'allocation (algorithme 1). Sachant que T_{CP} diminue quand on augmente le nombre de processeurs alloués à chaque tâche tandis que T_A augmente, on initialise les allocations en partant du cas où T_{CP} est maximal. On alloue ainsi un processeur à chaque tâche. Ensuite à chaque itération, on alloue un processeur de plus à la tâche la plus critique jusqu'à ce qu'on obtienne $T_{CP} \leq T_A$. Cette tâche critique est la tâche qui se trouve sur le chemin critique du graphe des tâches et dont le rapport $\frac{T_\omega(t, N_p(t))}{N_p(t)}$ diminue le plus significativement lorsqu'on lui alloue un processeur de plus. Dès qu'elle est vérifiée, la condition d'arrêt de l'algorithme ($T_{CP} \leq T_A$) traduit le fait que T_p^e sera très proche de sa valeur minimale ($T_p^e \approx T_{CP} \approx T_A$) après un ordonnancement adéquat.

À l'issue de cette phase d'allocation on associe donc à chaque tâche non seulement le nombre de processeurs qui lui seront alloués, mais aussi son *bottom level* qui servira de fonction de priorité pendant la phase d'ordonnancement.

Algorithm 1 MA : allocation de processeurs

procédure MA
  pour tout $t \in \nu$ **faire**
    $N_p(t) \leftarrow 1$;
  fin pour
  tant que $T_{CP} > T_A$ **faire**
    $t \leftarrow$ tâche du chemin critique / $(N_p(t) < P)$
    et $(\frac{T_\omega(t,N_p(t))}{N_p(t)} - \frac{T_\omega(t,N_p(t)+1)}{N_p(t)+1})$ est maximum ;
    $N_p(t) \leftarrow N_p(t) + 1$;
    Mettre à jour les T_t et T_b ;
  fin tant que
fin procédure

3.2 Ordonnancement des tâches

À chaque itération de la procédure d'ordonnancement, la tâche prête ayant le plus grand T_b, celle qui se trouve sur le chemin critique, est ordonnancée. Cette phase (algorithme 2) tient compte du coût de redistribution des données pour déterminer la date de début d'exécution, $T_s(t)$, et la date de fin d'exécution, $T_f(t)$, de chaque tâche t ordonnancée.

Algorithm 2 MLS : Ordonnancement de liste

procédure MLS
  tant que il reste des tâches non ordonnancées **faire**
    $t \leftarrow$ tâche prête ayant le $T_b(t)$ maximum.
    Ordonnancer t sur les $N_p(t)$ premiers processeurs libres.
  fin tant que
fin procédure

Nous allons maintenant voir comment adapter cet algorithme au cas des plates-formes hétérogènes.

4 Adaptation de l'algorithme CPA aux plates-formes hétérogènes

La plate-forme utilisée est maintenant constituée d'un ensemble hétérogène de grappes dans lequel chaque grappe est elle même constituée d'un ensemble homogène de processeurs.

Pour des questions de performance, on interdira à une même tâche de s'exécuter sur des processeurs appartenant à des grappes différentes. En effet, du fait de de l'existence de temps de latence importants entre grappes distinctes, il serait très coûteux en temps et en communication d'envisager des échanges de données entre deux grappes pour l'exécution d'une même tâche. Il faut ajouter à cela les problèmes de synchronisation entre des processeurs de différentes vitesses. Cette hypothèse permet également de simplifier la prédiction des temps d'exécution des tâches car celles-ci seront toujours placées sur des ensembles homogènes de processeurs. On pourra donc utiliser la loi d'Amdahl.

C étant le nombre de grappes, notre idée consiste à attribuer à chaque tâche, lors de la première

phase, une allocation de référence qui représente C allocations potentielles correspondant aux différentes grappes de la plate-forme. Une fonction que nous allons définir permettra de déterminer le nombre de processeurs alloués à une tâche en fonction de chaque grappe et de son allocation de référence.

La phase d'ordonnancement va ensuite permettre de choisir l'allocation qui minimisera le temps de fin d'exécution de la tâche prête la plus prioritaire.

4.1 Nouvelles définitions et Notations

Le tableau Tab. 2 présente les nouvelles notations qui seront utilisées dans la présentation de nos algorithmes.

Symbole	**Définition**
C	nombre de grappes
P_{ref}	nombre total de processeurs sur la grappe de référence
P_i	nombre total de processeurs sur la grappe C_i
${N_p}^i(t)$	nombre de processeurs alloués à la tâche t sur la grappe C_i
${T_\omega}^i(t, {N_p}^i(t))$	temps d'exécution de la tâche t sur ${N_p}^i(t)$ processeurs de la grappe C_i
r_i	rapport de la puissance d'un processeur de la grappe de référence sur celle d'un processeur de la grappe C_i

Tab. 2 – Liste des notations utilisées.

Étant donné que l'on a maintenant plusieurs allocations possibles pour une même tâche, il convient de redéfinir formellement la notion de chemin critique du graphe de tâches, T_{CP}, ainsi que celle du temps moyen d'occupation des processeurs, T_A, qui déterminent la condition d'arrêt de la phase d'allocation ($T_{CP} \leq T_A$) de CPA.

On introduit pour cela la notion de *grappe de référence* sur laquelle on fera évoluer l'allocation des processeurs. Cette *grappe de référence* est une plate-forme homogène virtuelle ayant une puissance de calcul cumulée équivalente à celle de l'ensemble des grappes de la plate-forme et dont les processeurs ont la même vitesse que ceux de la grappe composé des processeurs les plus lents.

Le nombre total de processeurs contenu dans la *grappe de référence* est donc :

$$P_{ref} = \left\lceil \sum_{i=0}^{C-1} \frac{P_i}{r_i} \right\rceil$$

où r_i est le rapport de la puissance d'un processeur de la grappe de référence sur celle d'un processeur de la grappe C_i.

Le but de son utilisation est de réduire la complexité du nouvel algorithme d'allocation en nous ramenant à une plate-forme homogène virtuelle. Les allocations ne se feront qu'en fonction de la grappe de référence et on note ${N_p}^{ref}(t)$ comme étant l'allocation de référence pour la tâche t, c'est-à-dire le nombre de processeurs qui lui seraient attribués sur la grappe de référence. L'allocation de référence est telle que le temps d'exécution effectif de chaque tâche t sera relativement proche du temps qu'elle mettrait sur la grappe de référence : ${T_\omega}^{ref}\left(t, {N_p}^{ref}(t)\right)$.

Ainsi pour une tâche t, le calcul de $T_b(t)$ ne tiendra compte que des temps d'exécution relatifs aux allocations de référence $({T_\omega}^{ref}(t', {N_p}^{ref}(t')))$ de toutes les tâches t' du graphe des tâches. On en déduit :

$$T_{CP} = \max_{t \in \nu} T_b(t).$$

De même on a :

$$T_A = \frac{1}{P_{ref}} \sum_{t \in \nu} \left({T_\omega}^{ref} \left(t, {N_p}^{ref}(t) \right) \times {N_p}^{ref}(t) \right).$$

4.2 Allocation de processeurs

Dans cette première phase, on ne tient pas compte du coût des communications entre les différentes tâches du graphe. En effet cette procédure à pour but de minimiser le chemin critique de l'application qui est fonction uniquement des temps d'exécution de chaque tâche. Les placements des tâches ne sont pas encore déterminés. Les coûts des communications seront pris en compte lors de la phase d'ordonnancement pour déterminer les dates de début et de fin d'exécution de chaque tâche.

À chaque itération de cette procédure, les tâches se verront attribuer des processeurs en fonction de leurs temps d'exécution de référence et des relations de précédence imposées par le graphe de tâches de l'application.

Pour éviter une boucle infinie dans la procédure, on définit la notion de *chemin critique saturé*. On dira que le chemin critique est saturé si les allocations de références sont telles qu'on ne peut plus rajouter des processeurs aux tâches qui se trouvent sur le chemin critique soit :

$$\forall i, P_i \leq \lceil {N_p}^{ref}(t) \times r_i \rceil.$$

Dans ce cas, le nombre de processeurs à allouer à une tâche du chemin critique sur chaque grappe C_i est le nombre total de processeurs présents sur cette grappe et on ne peut plus réduire les temps d'exécution des tâches de ce chemin en augmentant leurs allocations de référence. T_{CP} est donc minimum et on décide d'arrêter la phase d'allocation dès que cet état est atteint.

Pour déterminer le nombre de processeurs à allouer à une tâche sur une grappe en partant de l'allocation de référence, on utilise la loi d'Amdahl et deux possibilités s'offrent à nous. Une première approche consiste à négliger le coût de la parallélisation, ce qui revient à considérer que $\alpha = 0$. Ceci conduit à un speedup linéaire, le temps d'exécution décroît linéairement en fonction du nombre de processeurs alloués à la tâche. La seconde approche tient compte de la valeur exacte de α pour chaque tâche.

Les deux sections suivantes présentent, selon ces deux approches, l'évolution des allocations de processeurs sur les grappes en fonction de l'allocation de référence d'une tâche.

4.2.1 Speedups linéaires

Si $\alpha = 0$, le temps d'exécution d'une tâche est inversement proportionnel au nombre de processeurs qui lui sont attribués :

$${T_\omega}^i(t, {N_p}^i(t)) = \frac{{T_\omega}^i(t,1)}{{N_p}^i(t)}.$$

D'où pour deux grappes différentes C_i et C_j on a

$${T_\omega}^i(t, {N_p}^i(t)) = {T_\omega}^j(t, {N_p}^j(t)),$$

si et seulement si

$${N_p}^i(t) = \frac{{T_\omega}^i(t,1)}{{T_\omega}^j(t,1)} \cdot {N_p}^j(t).$$

En considérant la grappe de référence dans la relation précédente on obtient :

$$\begin{aligned} {N_p}^i(t) &= \frac{{T_\omega}^i(t,1)}{{T_\omega}^{ref}(t,1)} \cdot {N_p}^{ref}(t) \\ &= r_i \cdot {N_p}^{ref}(t). \end{aligned}$$

Par conséquent, si ${N_p}^{ref}(t)$ est le nombre de processeurs qui seraient attribués à la tâche t sur la grappe de référence, on lui alloue alors

$$ {N_p}^i(t) = \min\{P_i, \lceil {N_p}^{ref}(t) \times r_i \rceil\} \tag{1}$$

processeurs sur la grappe C_i afin d'approcher au mieux le temps d'exécution qu'on aurait eu sur la grappe de référence.

4.2.2 Speedups suivants la loi d'Amdahl

D'après la loi d'Amdahl on a :

$$ {T_\omega}^i(t, {N_p}^i(t)) = {T_\omega}^{ref}(t, {N_p}^{ref}(t)), $$

si et seulement si :

$$ \left(\alpha + \frac{1-\alpha}{{N_p}^i(t)}\right) \cdot {T_\omega}^i(t,1) = \left(\alpha + \frac{1-\alpha}{{N_p}^{ref}(t)}\right) \cdot {T_\omega}^{ref}(t,1). $$

Ce qui conduit à

$$\begin{aligned} {N_p}^i(t) &= \frac{(1-\alpha) \cdot {T_\omega}^i(t,1) \cdot {N_p}^{ref}(t)}{(1-\alpha) \cdot {T_\omega}^{ref}(t,1) + \alpha \cdot {N_p}^{ref}(t) \cdot ({T_\omega}^{ref}(t,1) - {T_\omega}^i(t,1))} \\ &= f({N_p}^{ref}(t), t, i) \end{aligned}$$

La fonction $f({N_p}^{ref}(t), t, i)$ ainsi définie nous permettra de déterminer le nombre de processeurs à allouer à chaque tâche t sur les différentes grappes de la plate-forme réelle en fonction de son allocation de référence :

$$ {N_p}^i(t) = \min\{P_i, \lceil f({N_p}^{ref}(t), t, i) \rceil\} \tag{2}$$

4.2.3 Comparaison des deux approches

Les figures 2 et 3 illustrent un exemple d'allocation de processeurs à une tâche t et l'évolution de son temps d'exécution en fonction de ${N_p}^{ref}(t)$ sur deux grappes en se basant sur les équations (1) et (2) avec :

- $\alpha(t) = 0.2$
- ${T_\omega}^0(t,1) = 1$
- ${T_\omega}^1(t,1) = 0.6$

On suppose que chaque grappe contient plus de 100 processeurs.

La figure 2 montre que, du fait de l'approximation $\alpha = 0$ dans la première approche, on note une différence non négligeable entre les différents temps d'exécution de la tâche et le temps d'exécution de référence. À l'inverse, en tenant compte du speedup issu de la loi d'Amdahl, on observe que

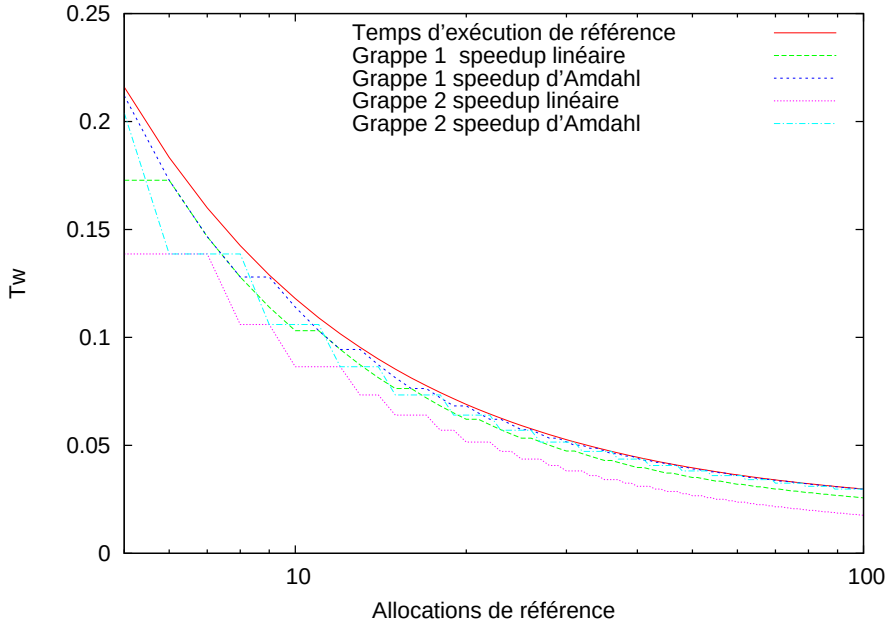


FIG. 2 – Évolution des temps d'exécution en fonction de ${N_p}^{ref}(t)$.

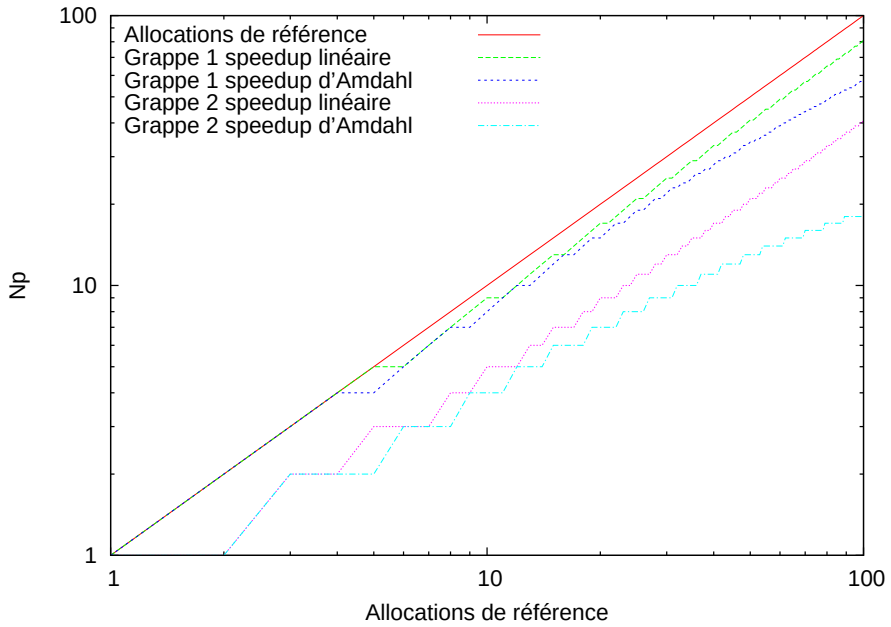


FIG. 3 – Évolution du nombre de processeurs en fonction de ${N_p}^{ref}(t)$.

les temps d'exécution de la tâche sont proches sur les deux grappes et convergent vers le temps d'exécution de référence. En effet la figure 3 montre que plus l'allocation de référence croît, plus on alloue d'avantage de processeurs qu'il n'en faut pour approcher le temps d'exécution de référence dans le cas où on ne tient pas compte de α.

Afin d'éviter des écarts importants entre les temps d'exécution dûs au facteur α, nous allons par la suite tenir compte des speedups définis par la loi d'Amdahl.

Il en découle la nouvelle version de l'algorithme d'allocation de processeurs (algorithme 3) adaptée aux plates-formes hétérogènes.

4.2.4 Algorithme d'allocation

Algorithm 3 Allocation de processeurs suivant le speedup de Amdahl

procédure AA
  pour tout $t \in \nu$ **faire**
    ${N_p}^{ref}(t) \leftarrow 1$;
    pour $i \leftarrow 0, C-1$ **faire**
      ${N_p}^{i}(t) \leftarrow 1$;
    fin pour
  fin pour
  tant que $T_{CP} > T_A$ et chemin critique non saturé **faire**
    $t \leftarrow$ tâche du chemin critique / $(\exists i / \lceil f({N_p}^{ref}(t), t, i) \rceil < P_i)$
    et $\left(\frac{T_\omega{}^{ref}(t, {N_p}^{ref}(t))}{{N_p}^{ref}(t)} - \frac{T_\omega{}^{ref}(t, {N_p}^{ref}(t)+1)}{{N_p}^{ref}(t)+1} \right)$ est maximum ;
    ${N_p}^{ref}(t) \leftarrow {N_p}^{ref}(t) + 1$;
    Mettre à jour les T_b ;
  fin tant que
fin procédure

4.3 Algorithmes d'ordonnancement

L'ordonnancement va maintenant consister à attribuer à chaque tâche prête t l'allocation qui lui garantit la plus petite échéance $T_f(t)$. Rappelons que l'allocation de référence fournit potentiellement C allocations distinctes où C est le nombre de grappes de la plate-forme.

Dans cette phase, on tient compte du coût des communications dues à la redistribution des données. On note $T_r^i(t_1, t_2)$ le coût de la redistribution des données lorsqu'on passe d'une tâche t_1 qui vient de s'exécuter sur un ensemble de processeurs connues à l'exécution d'une de ses tâches filles t_2 sur une grappe C_i.

$T_r^i(t_1, t_2)$ dépend de plusieurs facteurs dont les caractéristiques du réseau, la quantité de données à transférer, les nombres de processeurs alloués respectivement à la tâche prédécesseur t_1 et à la tâche successeur t_2.

On note $T_m^i(t)$ la date d'arrivée du dernier message d'une tâche prête t si celle-ci devrait s'exécuter sur la grappe C_i.

$$T_m^i(t) = \max_{t' \in \epsilon}(T_f(t') + T_r^i(t', t))$$

Dans ce cas, la date à laquelle la tâche t peut effectivement démarrer son exécution est

$$T_s^i(t) = \max\{dispo({N_p}^i(t)), T_m^i(t)\}$$

Où $dispo({N_p}^i(t))$ est la date à laquelle la grappe C_i aura au moins ${N_p}^i(t)$ processeurs libres

Les allocations potentielles des tâches étant prédéfinies grâce aux allocations de référence, nous avons le choix entre plusieurs algorithmes pour l'ordonnancement des tâches. Dans la section qui suit nous réadaptons l'algorithme d'ordonnancement de liste de CPA au cas où on dispose de plusieurs allocations possibles pour chaque tâche. La tâche la plus prioritaire pour l'ordonnancement est la tâche prête qui se trouve sur le chemin critique. La section 4.3.2 présente une deuxième approche dans laquelle la tâche la plus prioritaire pour l'ordonnancement est celle qui serait la plus pénalisée parmi les tâches prêtes si on ne lui attribue pas sa meilleure allocation.

4.3.1 Utilisation du *bottom level* comme critère de priorité

Ce critère est celui qui est utilisé par les concepteurs de CPA. Parmi toutes les tâches prêtes, la tâche la plus prioritaire est celle qui a le *bottom level* le plus élevé.

Une fois la tâche la plus prioritaire déterminée, nous allons choisir comme allocation celle qui minimise sa date de fin d'exécution. La date de fin d'exécution d'une tâche t est dans le meilleur des cas :

$$T_f(t) = \min_i(T_s^i(t) + {T_\omega}^i(t, {N_p}^i(t)))$$

On en déduit C_i, la grappe sur laquelle on prévoit l'exécution de la tâche t ainsi que la date de début d'exécution de t :

$$T_s(t) = T_s^i(t)$$

D'où le premier algorithme d'ordonnancement (Algorithme 4).

Algorithm 4 BS : Ordonnancement à l'aide du *bottom level*

```
1: procédure BS
2:     tant que il reste des tâches non ordonnancées faire
3:         t ← tâche prête ayant T_b(t) maximum
4:         pour i = 0 à i = C − 1 faire
5:             Calculer T_s^i(t)
6:         fin pour
7:         Calculer T_f(t), T_s(t) et Ordonnancer t
8:     fin tant que
9: fin procédure
```

4.3.2 Utilisation de l'heuristique *sufferage*

Le principe de l'heuristique *sufferage* [8] est de choisir parmi les tâches prêtes celle qui serait la plus pénalisée si on lui attribuait sa deuxième meilleure allocation au lieu de la première. Le critère d'optimisation dans le choix de l'allocation reste toujours la date de fin d'exécution des tâches.

On note alors

$$T_f^i(t) = T_s^i(t) + {T_\omega}^i(t, {N_p}^i(t))$$

la date de fin d'exécution d'une tâche prête t sur la grappe i.

Étant donnée une fonction g de $\mathbb{R}^n$ vers $\mathbb{R}$, on note argmin_x (respectivement argmax_x) le vecteur x qui minimise (respectivement maximise) $g(x)$.

Dans notre cas on prendra $g(i) = T_f^i(t)$ puis $g(t) = suf(t)$ où $suf(t)$ est la différence entre les temps de fin d'exécution d'une même tâche t lorsqu'on considère sa première et sa deuxième meilleure allocation.

Soit T l'ensemble des tâches prêtes. L'adaptation de *sufferage* à nos allocations conduit à un nouvel algorithme d'ordonnancement (algorithme 5).

Algorithm 5 SS : Ordonnancement avec *sufferage*

```
1: procédure SS
2:     tant que T ≠ ∅ faire
3:         pour chaque t ∈ T faire
4:             j(t) = argmin_i(T_f^i(t))
5:             k(t) = argmin_{i≠j(t)}(T_f^i(t))
6:             suf(t) = T_f^{k(t)}(t) − T_f^{j(t)}(t)
7:         fin pour
8:         t ← argmax_t(suf(t))
9:         Ordonnancer t sur la grappe j(t).
10:        T = T − {t}
11:    fin tant que
12: fin procédure
```

On obtient ainsi deux nouveaux algorithmes adaptés à l'ordonnancement de tâches parallèles sur plates-formes hétérogènes. Il s'agit de HCPA (Heterogenous CPA) qui résulte de l'addition des algorithmes 3 et 4 et de S-HCPA (Sufferage-based Heterogenous CPA) qui résulte quant à lui de l'addition des algorithmes 3 et 5.

4.4 Analyse de complexité

Soit $K = \max_{(i,t)} \left\lceil f^{-1}\left(P_i, t, i\right)\right\rceil$ le nombre de processeurs maximum qu'on pourrait attribuer à une tâche sur la grappe de référence. Dans le pire des cas on réaliserait K itérations pour chaque tâche dans la procédure d'allocation. D'où un total de $K \times V$ itérations. la complexité du corps de la boucle (choix du chemin critique, calcul de T_{CP}, T_b, et T_A) est de l'ordre de $O(V + E) \times C$. On en déduit la complexité de la phase d'allocation : $O\left(V + E \cdot C \cdot K \cdot V\right)$.

En ce qui concerne la phase d'ordonnancement, dans le cas de HCPA on trie les tâches par ordre de priorité à l'aide d'une application de l'ordre de $O(V + E)$, ensuite on teste les C allocations possibles de la tâche prête la plus prioritaire (au total $O(C \times V)$) avant de l'ordonnancer (au total $O(V \times K)$). On peut donc négliger la complexité de cette phase par rapport à la celle de la phase d'allocation. D'où une complexité globale de $O\left(V + E \cdot C \cdot K \cdot V\right)$ pour HCPA.

Si on considère maintenant la phase d'ordonnancement de SHCPA, dans le pire des cas on examine $V - i$ tâches prêtes à la i^{eme} itération. Avant d'ordonnancer la tâche la plus prioritaire, on evalue le temps d'éxécution de chaque tâche prête sur ces C allocations possibles. On en déduit que la complexité de la phase d'ordonnancement de SHCPA est de l'ordre de $C \times V^2$. D'où une complexité globale de $O\left(V + E \cdot C \cdot K \cdot V + C \times V^2\right)$ pour SHCPA.

5 Méthodologie d'évaluation

Afin d'évaluer les deux algorithmes que nous venons de concevoir, nous allons les comparer à d'autres algorithmes existants dans les mêmes conditions expérimentales. Pour pouvoir garantir la reproductibilité des conditions expérimentales, nous aurons recours à des simulations. Pour ce faire, nous utiliserons l'outil de simulation à événements discrets *SimGrid* [9] qui nous permettra de réaliser un très grand nombre de simulations. *SimGrid* est un outil adapté à la simulation des applications parallèles sur les plates-formes distribuées que sont les grilles de calcul.

Notre plan de test fixe des paramètres qui permettront de générer aléatoirement des grilles et les graphes de tâches. Les sections qui vont suivre présentent respectivement les grilles, les applications et les algorithmes que nous allons utiliser pour réaliser nos simulations.

5.1 Les grilles

On génère des plates-formes constituées de 1, 2, 4 ou 8 grappes de processeurs. Le nombre de processeurs de chaque grappe est tiré aléatoirement entre 16 et 128. Les grappes de la plate-forme sont homogènes et on alterne les caractéristiques des liens intra-grappes entre les deux valeurs suivantes :

- lien avec un débit de 100Mb/s pour une latence de 10^{-4} secondes (de type Fast Ethernet).
- lien avec un débit de 1Gb/s pour une latence de 10^{-4} secondes (de type Giga Ethernet).

Le backbone à un débit de 2.5 Gb/s et une latence de 50ms. Chaque passerelle a une capacité de 1 Gb/s avec une latence de 10^{-4}s. La capacité des passerelles limite le nombre de processeurs d'une même grappe qui émettent en même temps sur le réseau. Les routes TCP de la plate-forme sont fixées à l'avance et il n'existe qu'une route pour relier deux processeurs distincts de la plate-forme. La figure 4 montre que cette route, représentée par les flèches dans le schéma, est le plus court chemin entre les deux processeurs.

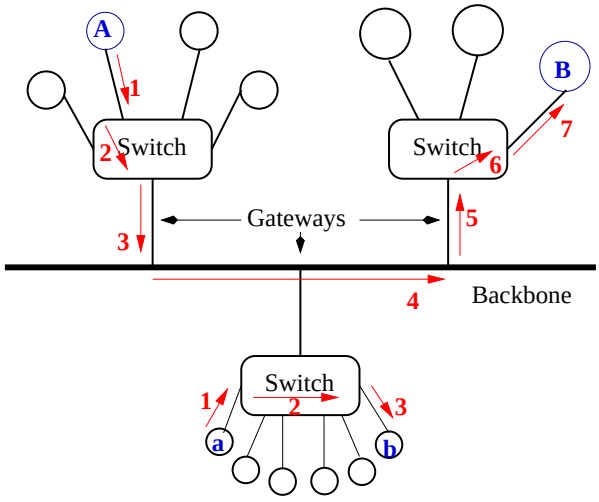


FIG. 4 – Exemples de routes TCP entre deux processeurs A et B situés dans deux grappes différentes et entre deux processeurs a et b situés dans la même grappes.

La borne inférieure des vitesses des processeurs de la grille en milliards d'opérations par seconde (GFlops), pourra quant à elle avoir comme valeur 0.25, 0.50, 0.75 ou 1. Ceci permet de faire varier le rapport entre la vitesse de calcul de la plate-forme et la vitesse de communication déjà fixée. Dès que l'on a plus de deux grappes, le rapport entre cette borne inférieure et la borne supérieure des vitesses

des processeurs de la plate-forme qui constitue son paramètre d'hétérogénéité sera choisi parmi les nombres 1, 2 ou 5. On choisi ainsi la vitesse des processeurs des différentes grappes de la plate-forme en tirant aléatoirement des vitesses entre les bornes inférieure et supérieure. Par exemple, si la borne inférieure est de 1GFlops et l'hétérogénéité est de 5, la vitesse des processeurs les plus lents sera au moins de 1GFlops et celle des processeurs les plus rapides sera au plus de 5GFlops. Le tableau TAB. 3 représente les différentes bornes supérieures (max_rel_speed) en fonction des bornes inférieures (min_rel_speed) et de l'hétérogénéité des plates-formes.

`min_rel_speed`		0.2	0.4	0.6	0.8	1
`max_rel_speed`	`het` $= 1$	0.2	0.4	0.6	0.8	1
	`het` $= 2$	0.4	0.8	1.2	0.8	2
	`het` $= 5$	1	2	3	4	5

TAB. 3 – Vitesse des processeurs (en Gflops).

Du fait des paramètres aléatoires, pour chaque cas on décide de générer 5 échantillons de grilles différentes. On génère ainsi 200 grilles différentes qui permettront de tester nos algorithmes. Le tableau TAB. 4 résume les paramètres de génération de grilles utilisés. Ainsi, la plus petite grille que nous avons générée contient une grappe de 21 processeurs alors que la plus grande est constituée de 8 grappes, pour un total de 828 processeurs.

nombre de grappes	1, 2, 4, 8
bornes inférieures des vitesses de processeurs (GFlops)	0.25, 0.5, 0.75 1
hétérogénéité	1, 2, 5
nombre minimum de processeurs par grappe	16
nombre maximum de processeurs par grappe	128
nombre d'échantillons	5

TAB. 4 – Paramètres de génération des grilles.

5.2 Les graphes de tâches

La figure 5 présente un exemple de graphe de tâches que l'on pourrait générer en utilisant les paramètres que nous allons décrire dans cette section. Les nœuds du graphe de l'application (les différentes tâches parallèles qui composent l'application) sont modélisés par des nœuds de calcul (nœuds 1, 2, 3, 8, 9 et 13) dans le simulateur. Les nœuds de transfert (nœuds 4, 5, 6, 7, 10, 11 et 12) représentent quand à eux les mouvements des données nécessaires à une tâche et générées par une autre tâche qui lui précède. Ces nœuds correspondent aux arrêtes du graphe de l'application. Il s'agit d'un graphe à trois niveaux comportant 6 tâches. Le nœud 12 illustre un exemple de saut de niveau dans le graphe de tâches. Les nœuds **Root** et **End** sont respectivement les tâches d'entrée et de sortie du graphe.

Les graphes de tâches sont générés en tirant n, un multiple de 1024 (1ko), entre les valeurs 2048 et 11268. n représente la taille des données que chaque tâche traite. En effet on suppose que chaque nœud de la grille a une capacité mémoire de 1 Go, ce qui limite la taille des matrices traitées à 11268×11268 (en considérant des éléments double précision). Ensuite selon le type de graphe de tâche que l'on désire, la complexité de chaque tâche est fixée : soit à $a \cdot N$, soit à $a \cdot NlogN$, soit à $a \cdot N^{3/2}$; soit tirée au sort parmi les trois valeurs précédentes. a est un nombre tiré aléatoirement

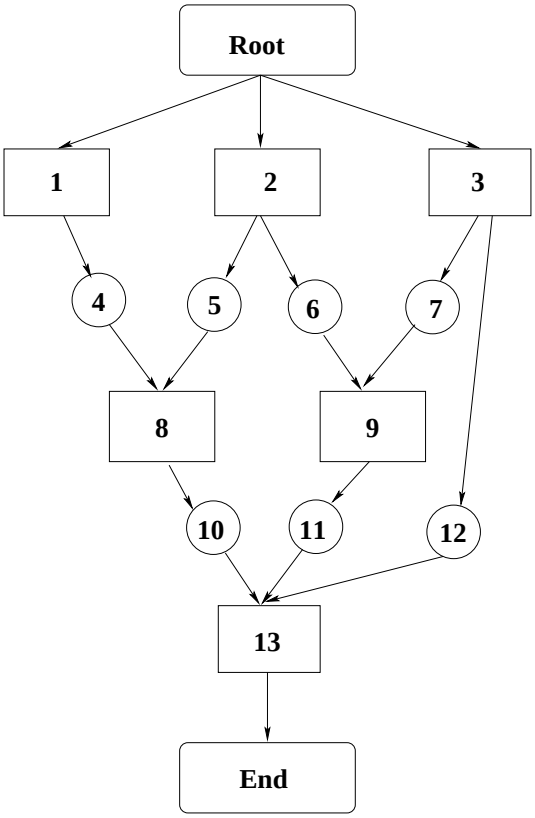


FIG. 5 – Exemple de graphe de tâches.

entre 2^6 et 2^9 et $N = n^2$. On assigne à chaque tâche de l'application sa portion non parallélisable α (loi d'Amdahl) en tirant aléatoirement un nombre entre 0 et 0.2. Ainsi il est possible de générer des tâches totalement parallèles ($\alpha = 0$). Le fait de limiter α à 0.2 traduit le fait qu'au plus 20% d'une tâche doit s'exécuter séquentiellement. Le coût des transferts est égal à N, où N est relatif à la tâche qui génère le transfert en question. La variation de la complexité permet donc de jouer sur l'importance relative des tâches de calcul par rapport aux tâches de transferts.

Quatre paramètres permettent de régler la forme du graphe de tâche. On peut ainsi préciser la largeur (ou aussi la profondeur) du graphe par rapport au nombre de tâches qu'il contient en choisissant un nombre parmi les valeurs 0.1, 0.2 et 0.8. Une largeur de 0.1 correspond plutôt à un graphe filiforme avec peu de parallélisme entre les tâches et une largeur de 0.8 correspond à un graphe compact avec beaucoup de parallélisme de tâches (voir figure 6). On peut également décrire le degré de régularité relatif au nombre de tâches à chaque niveau à l'aide des nombres 0.2 et 0.8. La figure 7 présente un exemple de graphe de régularité 0.1 où la différence entre les nombres de tâches sur deux niveaux consécutifs peu être très important et un exemple de graphe de régularité 0.8 où cette différence est moins importante. La densité qui caractérise le fait qu'on a plus ou moins de liaisons (relations de précédence) entre les tâches de l'application sera déterminée par le choix entre 0.2 et 0.8. L'effet de ce paramètre sur la génération des graphes est présenté sur la figure 8. On note sur cette figure qu'un graphe de densité 0.1 comprend un nombre minimal de liaisons entre les tâches tandis qu'un graphe de densité 0.8 contient beaucoup plus de liaisons. Enfin, on peut préciser la longueur des sauts entre niveaux qu'on autorise dans le graphe de tâche à générer (1, 2 et 4). Ce dernier paramètre sert à générer des graphes contenant des chemins de longueur différentes (en nombre de nœuds) de **Root** vers **End**.

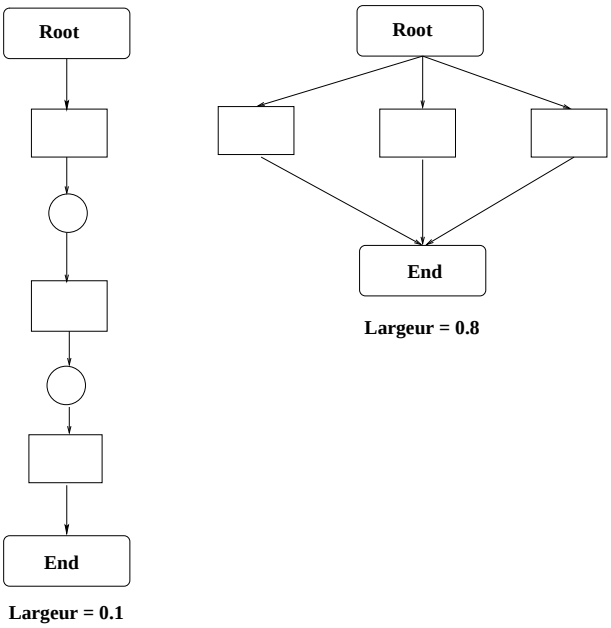


FIG. 6 – Variation de la largeur du graphe.

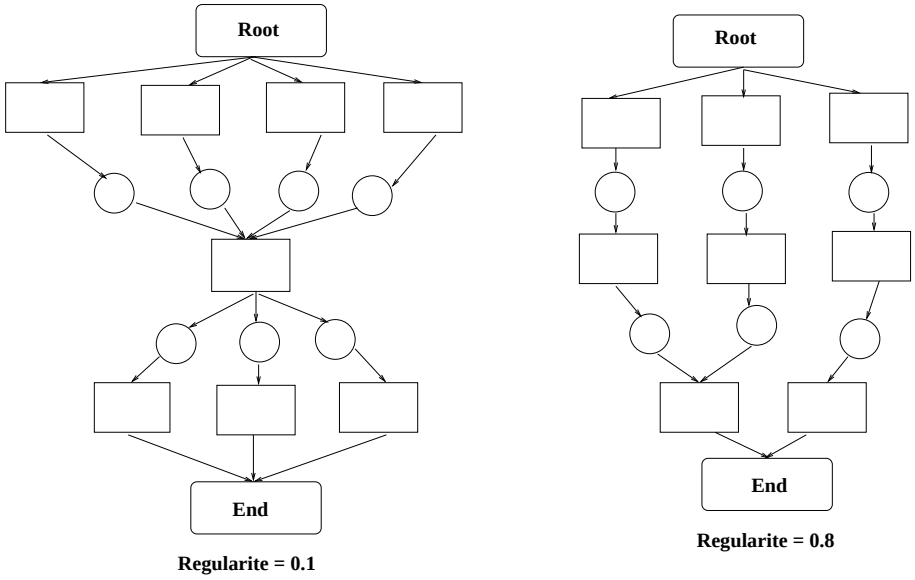


FIG. 7 – Variation de la régularité du graphe.

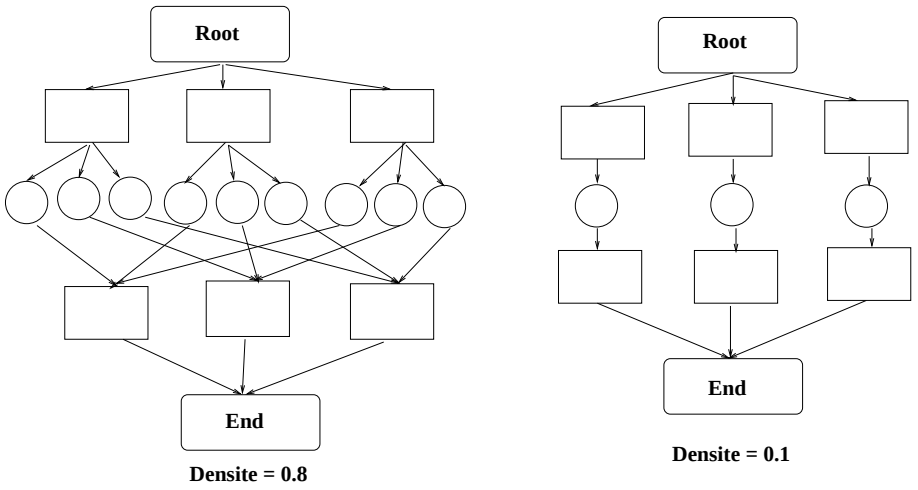


FIG. 8 – Variation de la densité du graphe.

Du fait des paramètres aléatoires, pour chaque cas, on décide de générer 3 échantillons de graphes différents. Le tableau TAB. 5 résume les paramètres de génération de graphes utilisés.

nombre de tâches de calcul	10, 20, 50
taille minimum des données	2048
taille maximum des données	11268
complexité des tâches de calcul	$a \cdot N$, $a \cdot NlogN$, $a \cdot N^{3/2}$, aléatoire
valeur minimale de α	0.0
valeur maximale de α	0.2
largeur des graphes	0.1, 0.2, 0.8
densité des graphes	0.2, 0.8
régularité entre les niveaux	0.2, 0.8
longueur des sauts	1, 2, 4
nombre d'échantillons	3

TAB. 5 – Paramètres de génération des graphes.

La combinaison de ces paramètres nous permet de générer au total 1296 graphes de tâches différentes.

5.3 Les algorithmes

Dans ces simulations, nous disposons de cinq algorithmes qui traiteront chacun des graphes générés sur chaque grille. D'où un total de 1 296 000 simulations à réaliser.

Ces algorithmes sont :

- **SEQ** : qui réalise l'ordonnancement séquentiel des tâches sur le processeur le plus rapide de la plate forme. La comparaison de SEQ à HCPA et CPA nous permettra d'observer le gain qu'on réalise en utilisant plus de processeurs.
- **CPA** : qui n'ordonnance les graphes que sur des plates-formes homogènes de processeurs. Afin de comparer **CPA** et nos deux algorithmes, pour chaque grille, on génère une grille homogène en termes de vitesse de processeurs sur laquelle ont exécutera **CPA**. Cette grille homogène garde

la structure de la grille d'origine (caractéristiques du réseau, nombre de grappes, nombre de processeurs contenus dans chaque grappe) mais les vitesses des processeurs sont maintenant la moyenne des vitesses de tous les processeurs de la plate-forme hétérogène dont elle dérive. La comparaison de cet algorithme avec HCPA et SHCPA permettra de prouver l'intérêt de ces derniers.

- **MHEFT** : algorithme d'ordonnancement de tâches parallèles sur plates-formes hétérogènes [7], adaptation aux tâches parallèles de l'heuristique d'ordonnancement séquentiel HEFT [10]. Les performances de cet algorithme seront comparées à celles de nos deux algorithmes.
- **HCPA et SHCPA** : nos algorithmes d'ordonnancement de tâches parallèles en milieu hétérogène que nous souhaitons évaluer.

6 Exploitation des résultats de simulations

Après avoir généré tout le plan de test que nous venons de décrire dans la section précédente, nous avons été confronté au fait que le simulateur *SimGrid* était inadapté à nos sollicitations. Ceci s'est traduit par une simulation qui se déroulait plus lentement que prévu. Ainsi au bout de quinze jours de simulation sans arrêt sur une trentaine d'ordinateurs, nous disposons seulement du cas des graphes de tâches à 10 tâches, des grilles à 1, 2 grappes et une partie des grilles à 4 grappes. On obtient tout de même environ 300 000 résultats de simulations qui nous permettent d'observer le comportement des différents algorithmes simulés.

La section 6.1 présentera les différentes métriques que nous allons utiliser pour vérifier les performances des algorithmes. Ensuite nous allons dans la section 6.2 comparer les performances de CPA, HCPA et SHCPA en milieu homogène avant l'étude des performances globales de nos deux nouveaux algorithmes. Cette comparaison permettra entre autre de vérifier si ces deux algorithmes donnent des résultats comparables à CPA sur les plates-formes homogènes à une grappe.

6.1 Métriques utilisés

Le *Makespan* (ou temps de complétion) d'une application est par définition la différence entre sa date de fin d'exécution et sa date de début d'exécution. Le but des algorithmes d'ordonnancement en deux étapes est de trouver le bon compromis entre le *Makespan* et la puissance de calcul utilisée pour exécuter une application. On essaye donc d'utiliser un minimum de puissance tout en essayant de minimiser le *Makespan* de l'application. Nous allons évaluer l'évolution de ce compromis en multipliant la moyenne des *Makespan* par la moyenne des pics de puissance obtenus dans les diverses simulations.

Une autre métrique intéressante consiste à définir l'efficacité de algorithmes d'ordonnancement de tâches parallèles par rapport à un algorithme d'ordonnancement séquentiel. Ainsi on définit le gain d'un algorithme noté ALG par rapport à l'algorithme SEQ par

$$Gain = \frac{Makespan \text{ de SEQ}}{Makespan \text{ de ALG}},$$

et l'efficacité de ALG par

$$Eff = \frac{Gain}{\text{Nombre de processeurs utilisés par ALG}}.$$

Nous aurons une idée de l'évolution de cette métrique en considérant comme nombre de processeurs utilisés, le nombre maximum de processeurs utilisés en même temps pendant l'exécution d'une application.

6.2 Comparaison de CPA, HCPA et SHCPA en milieu hétérogène

La figure 9 représente l'écart relatif entre les *Makespan* de HCPA par rapport à ceux de CPA quand on exécute les applications sur des plates-formes homogènes constituées d'une seule grappe. On relève que dans 49.05% des cas, le *Makespan* fournit par CPA est supérieur à celui de CPA. Les deux *Makespan* sont égaux dans 26.63% des cas et ceux de HCPA sont supérieurs dans 24.31% des cas. Toutefois on note que ces écarts sont importants que pour les faibles valeurs du *Makespan*. Ainsi les figures 10(a), (b) et (c) révèlent qu'on aboutit en moyenne à des performances similaires pour les deux algorithmes sur les plates-formes homogènes à une grappe. On note sur la figure 10(a) que les valeurs moyennes des *Makespan* de CPA, HCPA et SHCPA diminuent quand on tient compte de la totalité des plates-formes homogènes en termes de vitesse de processeurs (plates-formes d'hétérogénéité égale à 1). Ce qui est tout à fait normal car on considère maintenant les plates-formes composées d'un nombre plus important de processeurs. On vérifie alors que dans ce cas, le compromis moyenne des *Makespan* × moyenne des pics de puissance des trois algorithmes se dégrade (augmente) –voir figure 10(b)–.

On observe également que l'efficacité de CPA est deux fois moins importante que celui de HCPA lorsque l'on tient compte de toutes les plates-formes homogènes en termes de vitesse de processeurs. En effet, dans le cas de CPA qui n'interdit pas l'exécution d'une même tâche sur plusieurs grappes, l'hétérogénéité du réseau fait que le coût des communications devient très important lorsqu'on alloue des processeurs appartenant à des grappes différentes à une tâche. Ainsi on alloue beaucoup plus de processeurs aux tâches sans vraiment améliorer les *Makespan* par rapport à HCPA. Nous confirmerons le fait que CPA est complètement inadapté aux plates-formes hétérogènes dans la section 6.3.

D'autre part on note que SHCPA aboutit à des performances légèrement meilleures à celles de CPA dans le cas des plates-formes homogènes que nous venons de considérer. Le principe de l'heuristique *sufferage* selon lequel on évite le moins possible de pénaliser l'ensemble des tâches prêtes, est sans doute mieux adapté aux plates-formes homogènes en termes de vitesse de processeurs.

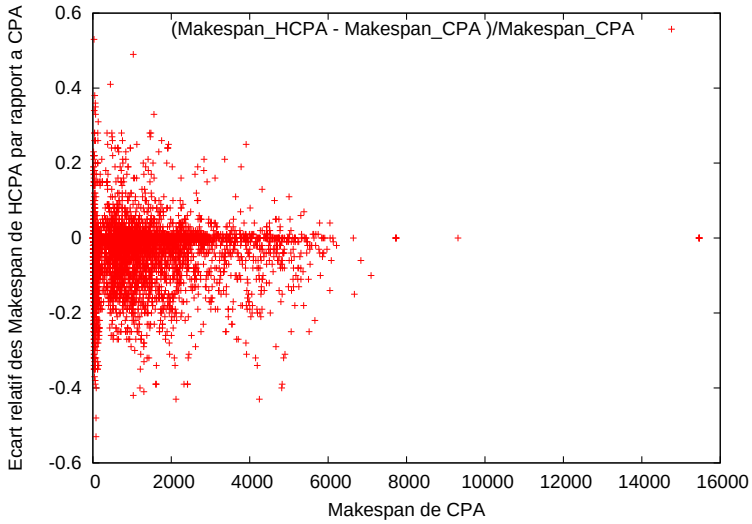
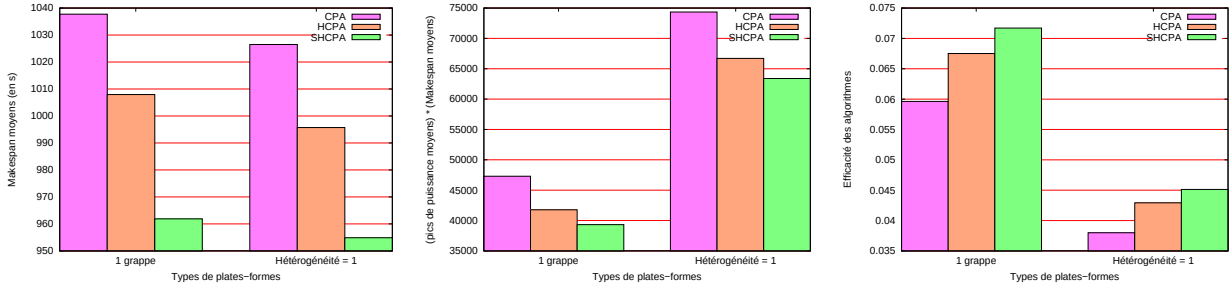


FIG. 9 – Écarts relatifs entre les *Makespan* de HCPA et ceux de CPA sur les plates-formes homogènes à une grappe.

Pour conclure, nous pouvons affirmer que nos deux nouveaux algorithmes peuvent être présentés comme une extension de CPA aux plates-formes hétérogènes car ils aboutissent à des performances



(a) Comparaison des *Makespan*

(b) Compromis *Makespan* × puissances

(c) Évolution de l'efficacité des algorithmes

FIG. 10 – Comparaison des performances de CPA, HCPA et SHCPA sur les plates-formes homogènes.

similaires, sinon meilleures, par rapport à celles de CPA sur les plates-formes homogènes à une grappe.

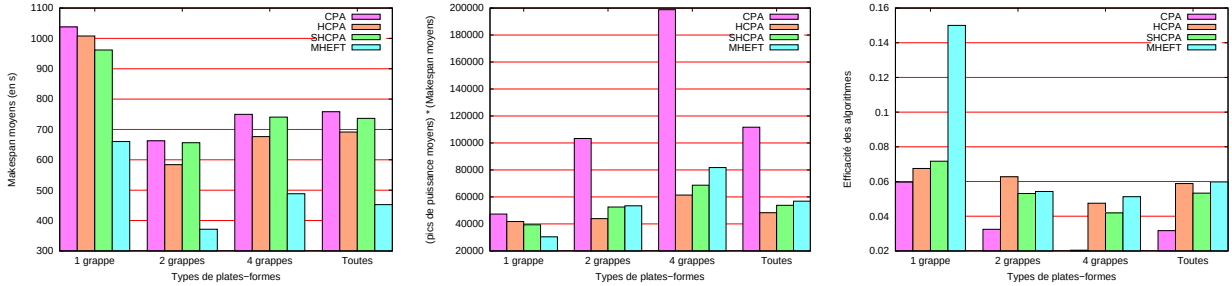
6.3 Performances de HCPA et SHCPA

Dans la figure 11(b) nous évaluons l'évolution du compromis moyenne des *Makespan* × moyenne des pics de puissance obtenus dans les diverses simulations. On observe que pour CPA, ce critère se dégrade et devient de plus en plus grand quand on augmente le nombre de grappes. Il en est de même pour son efficacité qui diminue fortement dès qu'on passe d'une grappe à plusieurs grappes. Or, toutes les simulations réalisées avec CPA ont été faites en homogénéisant les plates-formes en termes de vitesse de processeurs. Ces observations confirment le fait que CPA n'est pas adapté aux plates-formes constituées de plusieurs grappes même si celle-ci comportent des processeurs ayant tous les mêmes caractéristiques.

La figure 11(a) montre que l'heuristique MHEFT donne quant à lui, de meilleurs *Makespan* comparativement à ceux de HCPA et de SHCPA. Par contre la figure 11(b) nous laisse entrevoir que le compromis *Makespan* × *puissance utilisée* est moins bon pour MHEFT par rapport à nos deux algorithmes. On en déduit que le principe d'ordonnancement en deux étapes de HCPA et SHCPA permet d'obtenir de bons compromis entre le *Makespan* des applications et la puissance utilisée sur les plates-formes. En effet, la phase d'allocation des algorithmes HCPA et SHCPA attribut un maximum de processeurs uniquement aux tâches du chemin critique. Ceci n'est pas le cas pour MHEFT qui alloue un maximum de processeurs à chaque tâche. On remarque dans la figure 11(c) que, du fait de son faible *Makespan*, l'efficacité de MHEFT est maximale pour les plates-formes à une grappe. Mais cette efficacité diminue très vite quand le nombre de grappes des plates-formes augmente et devient comparable à celle de HCPA et SHCPA. Or cette efficacité diminue très peu avec HCPA et SHCPA qui parviennent à baisser les *Makespan* sans utiliser un nombre trop important de processeurs à la fois. Car les tâches moins prioritaires utilisent peu de processeurs.

On note de manière évidente que plus le nombre de grappes augmente, plus le compromis *Makespan* × puissance se dégrade pour tous les algorithmes (voir figure 11(b)). En effet, on sollicite de plus en plus le réseau qui est un facteur limitant dans l'amélioration du *Makespan*.

Les figures 12(a) et (c) montrent respectivement que les *Makespan* et efficacités de HCPA et de SHCPA croissent avec l'hétérogénéité des grappes. En effet on dispose de processeurs de plus en plus rapides pour les tâches les plus prioritaires. Du fait de la prise en compte des plates-



(a) Comparaison des *Makespan*

(b) Compromis *Makespan* × puissances

(c) Évolution de l'efficacité des algorithmes

FIG. 11 – Performances des algorithmes en fonction du nombre de grappes des plates-formes.

formes homogènes à une grappe, on remarque que l'efficacité de ces deux algorithmes est moins bonne pour les plates-formes d'hétérogénéité 1 par rapport à MHEFT. Mais l'efficacité de HCPA devient rapidement meilleure par rapport à MHEFT qui utilise toujours plus de processeurs quand on augmente l'hétérogénéité des plates-formes. La figure 12(b) confirme le fait que le compromis *Makespan* × puissance utilisée est meilleur pour HCPA par rapport à MHEFT sur les plates-formes hétérogènes. On remarque également que plus le réseau est hétérogène, plus le compromis *Makespan* × puissance est faible pour tous les algorithmes d'ordonnancement de tâches parallèles que nous avons simulés. Ceci peut être interprêté par le fait que l'on utilise principalement les processeurs les plus rapides au dépend des processeurs les plus lents et que, en définitive, cette sélectivité se traduit par l'utilisation d'un nombre moins important de processeurs.

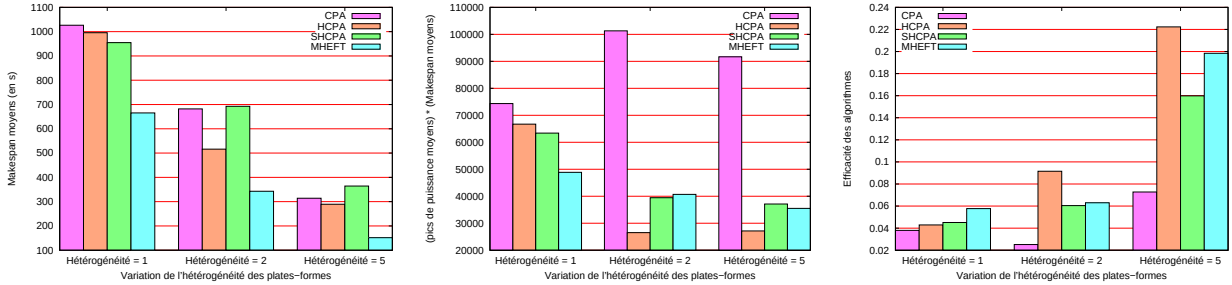
D'autre part on remarque que, parce qu'il ne tient pas compte du chemin critique dans l'ordonnancement des tâches, SHCPA fournit de moins bons résultats par rapport à HCPA sur les plates-formes fortement hétérogènes (voir figures 12(a), (b), (c) et 11(a), (b) et (c)). La figure 13 présente les écarts relatifs entre les *Makespan* de SHCPA par rapport à ceux de HCPA sur l'ensemble des plates-formes simulées. Dans 35.11% des cas, le *Makespan* fournit par HCPA est supérieur à celui de SHCPA. Les deux *Makespan* sont égaux dans 29.96% des cas et ceux de SHCPA sont supérieurs dans 34.79% des cas. Ceci nous permet d'affirmer que les heuristiques d'ordonnancement proposés pour HCPA et SHCPA peuvent encore être améliorées. Toutefois, la figure 11(a) montre qu'en moyenne le *Makespan* obtenu en utilisant HCPA est supérieur à celui de SHCPA.

On observe dans les figures 14(a) et (c) que l'on améliore les performances de HCPA, SHCPA et MHEFT quand la vitesse des processeurs est relativement importante par rapport à la vitesse du réseau. HCPA gérant mieux l'utilisation des processeurs par rapport MHEFT, voit son efficacité devenir plus importante que ce dernier. Cependant cette variation de vitesses n'influe pas sur le compromis *Makespan* × puissance (voir figure 14(b)).

Nous allons maintenant observer le comportement des algorithmes selon les caractéristiques des graphes de tâches qu'on leur soumet.

6.4 Comportement selon les caractéristiques des graphes de tâches

Les figures 15(b) et (c) montrent que la variation de la largeur des graphe n'influe pas sur les performances de HCPA, SHCPA et MHEFT. Cependant on note que pour les graphes plus



(a) Comparaison des *Makespan*

(b) Compromis *Makespan* × puissances

(c) Évolution de l'efficacité des algorithmes

FIG. 12 – Comportement selon l'hétérogénéité du réseau.

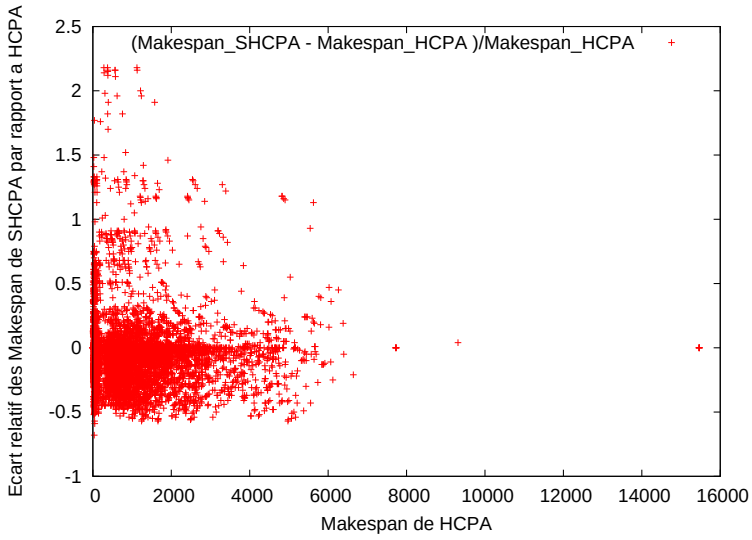
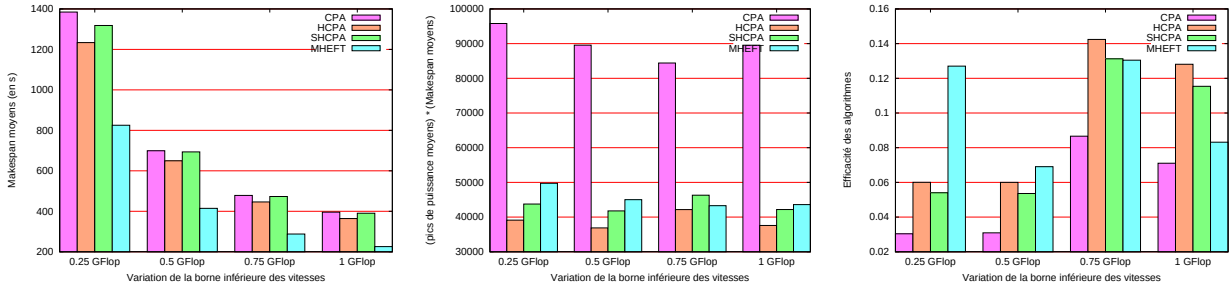


FIG. 13 – Écarts relatifs entre les *Makespan* de SHCPA et HCPA.



(a) Comparaison des *Makespan*

(b) Compromis *Makespan* × puissances

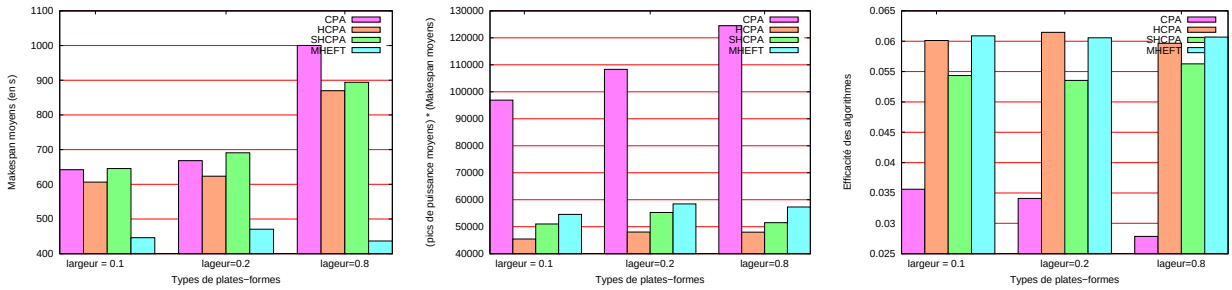
(c) Évolution de l'efficacité des algorithmes

FIG. 14 – Comportement selon le rapport entre la vitesse des processeurs et la vitesse du réseau.

larges (largeur = 0.8 dans la figure 15(a)) le *Makespan* devient important sans doute à cause des communications dues au parallélisme de tâches.

Les figures 16(a), (b) et (c) montrent que pour une faible complexité pour les tâches de calcul (complexité en N), on obtient de très bons *Makespan* pour un bon compromis *Makespan* $\times$ puissance avec tous les algorithmes. En effet l'exécution des tâches se fait plus rapidement. À l'inverse, on observe que l'efficacité de HCPA et SHCPA est moins bonne dans ce cas. En effet les communications réseaux deviennent relativement importantes. Or on ne tient pas compte de ces communications pendant la phase d'allocation de ces deux algorithmes. Or la prise des communications aurait pu éviter d'utiliser plus de processeurs et donc permis de limiter les communications inter-grappes.

Enfin, les figures 17(a), (b) et (b) montrent que SHCPA, par rapport à HCPA, est moins adapté à l'ordonnancement des tâches des graphes comportant des sauts entre niveaux. En effet ces graphes nécessitent que l'on tienne compte des chemins critiques pour pouvoir minimiser leur *Makespan*. D'autres observations nous ont permis de voir que la régularité et la densité des graphes de tâches n'ont pas d'influence sur la performance des algorithmes étudiés.

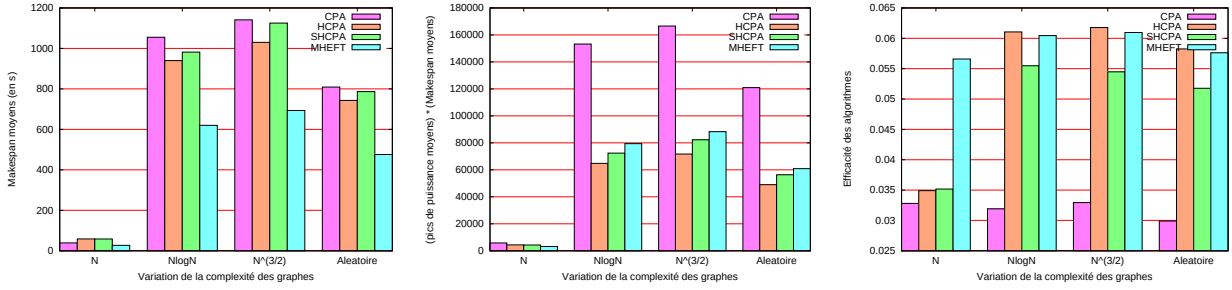


(a) Comparaison des *Makespan*

(b) Compromis *Makespan* $\times$ puissances

(c) Évolution de l'efficacité des algorithmes

FIG. 15 – Comportement selon la largeur des graphes.

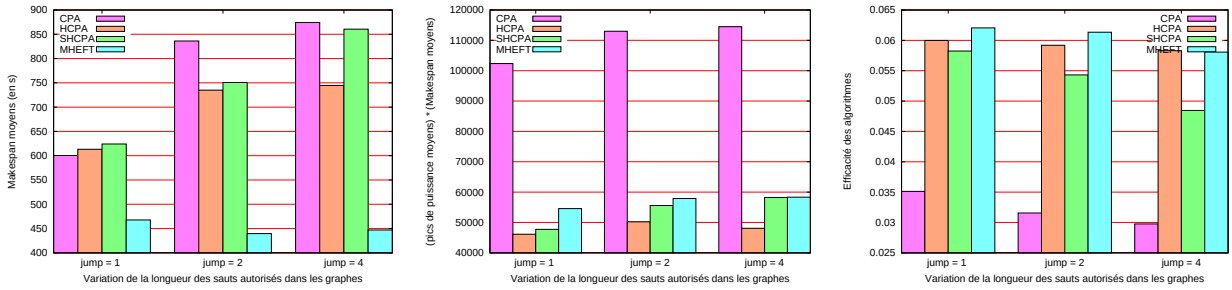


(a) Comparaison des *Makespan*

(b) Compromis *Makespan* $\times$ puissances

(c) Évolution de l'efficacité des algorithmes

FIG. 16 – Comportement selon la complexité des tâches de calcul.



(a) Comparaison des *Makespan*

(b) Compromis *Makespan* × puissances

(c) Évolution de l'efficacité des algorithmes

FIG. 17 – Comportement selon les longueurs de sauts autorisés.

Conclusion

Cette étude nous a permis d'adapter un algorithme d'ordonnancement de tâche parallèles sur plates-formes homogènes, CPA [4] aux plates-formes hétérogènes. Tout en étant adaptés aux milieux hétérogènes les algorithmes que nous avons mis en place, HCPA et CPA, permettent d'obtenir des résultats proches voire meilleurs que ceux de CPA quand on leur soumet des applications à exécuter sur les plates-formes homogènes. Le fait d'avoir conservé le principe qui consiste à dissocier les allocations de processeurs aux tâches et leur ordonnancement permet d'avoir un meilleur compromis entre le *Makespan* et la puissance utilisée par rapport à MHEFT surtout quand la complexité des tâches est relativement importante par rapport aux données échangées. En effet seuls les tâches critiques se voient allouer un grand nombre de processeurs dans les deux nouveaux algorithmes.

Les simulations que nous avons lancées nous ont permis d'avoir un premier aperçu du comportement des algorithmes. Malheureusement, au bout de deux semaines de simulations sur une trentaine d'ordinateurs, nous n'avons pas eu le temps de simuler les applications de grandes tailles comportant plus de 20 nœuds. Ils nous reste donc à lancer ces dernières simulations avant de conclure définitivement quant à la performance de HCPA et SHCPA face à MHEFT. À cette occasion, nous essayerons d'intégrer la gestion de notre modèle de tâches parallèles directement dans la bibliothèque *SimGrid*. Ceci nous permettra de gagner du temps avec *SimGrid* qui n'était pas complètement adapté à nos besoins de simulation.

L'étude que nous venons de réaliser supposait qu'on dispose de l'intégralité des ressources de la plate-forme utilisée. Or les grilles de calcul sont intrinsèquement partagées et permettent l'exécution concurrente de nombreuses applications. Un des aspects du sujet de thèse qui découle de ce DEA est de tenir explicitement compte du caractère partagé des grilles de calcul dans la conception des algorithmes d'ordonnancement de tâches parallèles.

Références

[1] Pierre-François Dutot, Lionel Eyraud, Grégory Mounié, and Denis Trystram. Models for scheduling on large scale platforms : Which policy for which application ? In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[2] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Transactions on Parallel Distributed Systems*, 8(11) :1098–1116, 1997.

[3] Andrei Radulescu, Cristina Nicolescu, Arjan van Gemund, and Pieter Jonker. CPR : Mixed task and data parallel scheduling for distributed systems. In *15-th International Parallel and Distributed Processing Symposium (IPDPS)*, apr 2001. Best Paper Award.

[4] Andrei Radulescu and Arjan van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *Proceedings of International Conference on Parallel Processing*, pages 69 –76, September 2001.

[5] Thomas Rauber and Gudula Rünger. Compiler support for task scheduling in hierarchical execution models. *Journal of System Architectures*, 45(6-7) :483–503, 1999.

[6] Vincent Boudet, Frédéric Desprez, and Frédéric Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[7] Henri Casanova, Frédéric Desprez, and Frédéric Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference (Euro-Par'04)*, volume 3149, pages 230–237, Pisa, Italy, August/September 2004. Springer.

[8] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov, and Francine Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Heterogeneous Computing Workshop*, pages 349–363, 2000.

[9] Henri Casanova, Arnaud Legrand, and Loris Marchal. Scheduling distributed applications : the simgrid simulation framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan*, pages 138–145, May 2003.

[10] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel Distributed Systems*, 13(3) :260–274, 2002.